CompSci310/ECE353 Midterm Exam (50 minutes), 09/29/2021

Name: NetID:

1. (5 points) Which of the following creates a binary from multiple object files?

   ◯ compiler ◯ assembler ◯ linker ◯ interpreter

2. (5 points) Giving each process an illusion of (almost) infinite amount memory corresponds to which role of the operating systems?

   ◯ Referee ◯ Illusionist ◯ Glue ◯ Resource manager

3. (3 points) When a process divides 1 by 0, what will happen next?

   ◯ CPU detects this exception ◯ OS kernel detects this exception

4. (5 points) Choose a circumstance that will NOT trigger a user to kernel transition?

   ◯ Exception ◯ Upcall ◯ Interrupt ◯ System call

5. (5 points) Why DMA is more efficient than programmed I/O?

   ◯ DMA saves CPU cycles.

   ◯ DMA reduces memory latencies.

   ◯ DMA reduces device-to-CPU communication latency.

   ◯ DMA does not require device to communicate with CPU through ports.

6. (5 points) One approach to provide virtual memory is to use base and bound: each process is only allowed to access a region of the physical memory, and the region is specified as a base and a bound. For example, if base = 5 and bound = 10, the process is only allow to access memory between 5 and 15. What's NOT an disadvantage of this approach?

   ◯ Expanding the heap is difficult

   ◯ Expanding the stack is difficult

   ◯ Memory sharing is difficult

   ◯ Memory isolation is difficult

7. (3 points) Virtual memory requires hardware support.

   ◯ True. ◯ False.

8. (5 points) The screen is 80 × 25 in size. Each pixel requires 2 bytes. The first byte is the ASCII code for the character and the second byte tells what color to draw the character. You know that the video buffer starts at `VIDEO_MEMORY` (type is void*) and the memory is organized line-by-line. Which of the following is the correct implementation of `putchar`?

   ◯
   ```
   void putchar(char val, int row, int col) {
           char* output_pos = VIDEO_MEMORY + row * 80 + col;
           *output_pos = val;
           *(output_pos + 1) = 0x7;
   }
   ```

   ◯
   ```
   void putchar(char val, int row, int col) {
           char* output_pos = VIDEO_MEMORY + row * 80 + col * 25;
           *output_pos = val;
           *(output_pos + 1) = 0x7;
   }
   ```

◯
```
void putchar(char val, int row, int col) {
        char* output_pos = VIDEO_MEMORY + row * 25 + col;
        *output_pos = val;
        *(output_pos + 1) = 0x7;
}
```

◯
```
void putchar(char val, int row, int col) {
        char* output_pos = VIDEO_MEMORY + row * 160 + col * 2;
        *output_pos = val;
        *(output_pos + 1) = 0x7;
}
```

9. (3 points) In the first programming assignment, you are asked to write code to read data from disk to memory.

```
uint8_t binary_to_load[SECTSIZE*MAX_RW];
ide_read(sec, binary_to_load, MAX_RW);
```

Is `binary_to_load` a virtual memory address or a physical memory address?

◯ virtual address ◯ physical address

10. (3 points) An ELF file contains multiple segments. Each segment contains an address of where the segment should be loaded into the memory. What type of address is it?

◯ virtual address ◯ physical address

11. (5 points) In the first assignment, you are asked to implement a simple ELF program loader to put the user code into correct memory region. You are given each of the program header `ph[i]` in a loop. Which one is the correct implementation of `void load_code(uint8_t* binary)`?

This is the ELF program header for reference.

```
typedef struct {
  ...
  uint32_t  p_offset;
  uint32_t  p_vaddr;
  uint32_t  p_paddr;
  uint32_t  p_filesz;
  uint32_t  p_memsz;
  ...
} Elf32_Phdr;
```

◯
```
void load_code(uint8_t* binary) {
    ...
    memcpy((void *) ph[i].p_va, (const void *) ph[i].p_filesz[binary +
    ↪  ph[i].p_offset], ph[i].p_memsz);
    // memset remaining bytes in the segment to 0
    ...
}
```

◯
```
void load_code(uint8_t* binary) {
    ...
    memcpy((void *)&ph[i].p_va, (const void *) (bindary + ph[i].p_offset),
    ↪  ph[i].p_filesz);
    // memset remaining bytes in the segment to 0
    ...
}
```

◯
```
void load_code(uint8_t* binary) {
  ...
  memcpy((void *) ph[i].p_va, binary + ph[i].p_offset, ph[i].p_filesz);
  // memset remaining bytes in the segment to 0
  ...
}
```

◯
```
void load_code(uint8_t* binary) {
  ...
  memset(binary + ph[i].p_va, 0, ph[i].p_memsz);
  // memset remaining bytes in the segment to 0
  ...
}
```

12. (3 points) An open file descriptor in a Unix process is an integer.

    ◯ True. ◯ False.

13. (5 points) How many times does the following program print "Hello!"?

```
int a = 0;
int rc = fork();
a++;
if (rc == 0) {
  rc = fork();
  a++;
} else {
  a++;
}
printf("Hello!\n");
printf("a is %d\n", a);
```

    ◯ 2 ◯ 3 ◯ 4 ◯ 6 ◯ None of the above

14. (5 points) Considering the same program, what will be the largest value of a, displayed by the program?

    ◯ 2 ◯ 3 ◯ 5 ◯ None of the above

15. (5 points) A buddy memory allocator is an approximation of

    ◯ first fit ◯ last fit ◯ best fit ◯ worst fit

16. (3 points) The heap manager (programming assignment #2) is a

    ◯ User-level library ◯ OS kernel service

17. (5 points) In the second assignment, you have implemented a heap manager. The idea is to divide the heap region into many aligned blocks. Each block is prepended with a header (`metadata_t`). Now the user wants to allocate `numbytes` of data, and your algorithm decides to split a new block from the block represented by `metadata_t* header`.

    The definition of `metadata_t` is given below.

```
typedef struct metadata {
  size_t size;
  struct metadata* next;
  struct metadata* prev;
} metadata_t;
/* ALIGN() rounds up to the nearest multiple of 8 */
#define METADATA_T_ALIGNED (ALIGN(sizeof(metadata_t)))
```

Which of the following is the correct implementation to get the address of the new header.

◯ `metadata_t* new_header = (metadata_t*)(header + METADATA_T_ALIGNED + ALIGN(numbytes));`

◯ `metadata_t* new_header = (void*)((size_t)header + METADATA_T_ALIGNED + ALIGN(numbytes));`

◯ `metadata_t* new_header = (void*)(header + 1);`

◯ `metadata_t* new_header = (metadata_t*)(header + 1 + numbytes);`

18. (5 points) Threads of the same process do NOT share:

◯ Code ◯ Global variables ◯ Heap ◯ Stack

19. (5 points) Consider the following solution proposed by your classmate to the Too Much Milk Problem. You can assume there is no instruction reordering. `note` is a global variable shared between A and B, and it is initialized to 0. Safety means it should not be the case that both A and B purchase milk. Liveness means as long as one of threads exists, there is milk eventually. This solution is:

```
Thread A
if (note == 0) {
  if (milk == 0) {
    buy_milk();
  }
  note = 1;
}
```

```
Thread B
if (note == 1) {
  if (milk == 0) {
    buy_milk();
  }
  note = 0;
}
```

◯ correct ◯ does not ensure safety
◯ does not ensure liveness ◯ does not ensure either safety or liveness

20. (5 points) Considering the same question but for three roommates. You can assume there is no instruction reordering. Initially, noteA = noteB = noteC = 0. This solution is:

```
Thread A
noteA = 1
while (noteB == 1 or
       noteC == 1) { }
if (milk == 0) {
  buy_milk();
}
noteA = 0
```

```
Thread B
noteB = 1
if (noteA == 0 and noteC == 0)
{
  if (milk == 0) {
    buy_milk();
  }
}
noteB = 0
```

```
Thread C
noteC = 1
if (noteA == 0 and noteB == 0)
{
  if (milk == 0) {
    buy_milk();
  }
}
noteC = 0
```

◯ correct ◯ does not ensure safety
◯ does not ensure liveness ◯ does not ensure either safety or liveness

21. (3 points) User-level applications need to turn off/on hardware interrupts to create critical sections.

◯ True. ◯ False.

22. (6 points) Using an R/W lock implementation (discussed in class), your classmate wrote the following code to protect the access to some shared data structure:

```
rwlock.acquire_read(); // A
// read the shared data structure
rwlock.release_read(); // B
rwlock.acquire_write(); // C
// modify the shared data structure
rwlock.release_write(); // D
```

You classmate know that inside the implementation of rwlock, there is a mutex. If you forget what's dicussed in the class, just think of a simplest implementation of r/w lock using mutex and condition variable. How many times does each line of the above code acquire the mutex? ($\geq$1 means at least once, but maybe more due to different thread schedules. 1 means exactly once in all thread schedules.)

- ◯ A=1, B=1, C=1, D=1
- ◯ A$\geq$1, B=1, C$\geq$1, D=1
- ◯ A=1, B=1, C$\geq$1, D$\geq$1
- ◯ A$\geq$1, B$\geq$1, C$\geq$1, D$\geq$1

23. (3 points) After return from a call to `cv_wait(cv, lock)` the calling thread can rely that the lock is locked.

◯ True. ◯ False.

24. (3 points) BONUS QUESTION: Consider the following two threads, to be run concurrently in a shared memory (all variables are shared between the two threads). No instruction reordering. Initially, x = 0.

```
Thread A
for (i=0;i<5;i++)
    x += 1;
```

```
Thread B
for (j=0;j<5;j++)
    x += 1;
```

Assuming a single-processor system, that load and store are atomic, that x is initialized to 0, and that x must be loaded into a register before being incremented (and stored back to memory afterwards), what is the MINIMUM POSSIBLE value for x after both threads have completed?

◯ 0 ◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ 5 ◯ 6 ◯ 7 ◯ 8 ◯ 9 ◯ 10